

1、扩展运算符：

扩展运算符（spread）也是三个点（...），它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列，对数组进行解包。

- 主要用于函数调用
- 扩展运算符 后可以放表达式
- 如果扩展运算符后面是一个空数组，则不产生任何效果
- 可以合并数组
- 可以与结构赋值结合起来生成数组
- 若扩展运算符用于数组，只能放在参数的最后一位，否则报错
- 可以将字符串转为真正的数组

```
1 let arr = [1, 2, 3];
2 console.log(...arr);//[1, 2, 3]
3 console.log(1, 2, 3);//[1, 2, 3]
4
5 //复制一个新的数组
6 let fruit = ['a', 'p', 'b', 'o'];
7 let newArr = [...fruit]; //复制 指向同一个空间
8 console.log(newArr);//[ 'a', 'p', 'b', 'o' ]
9 console.log(fruit === newArr);//false
10
11 //解构赋值
12 let [a, b, ...c] = [1, 2, 3, 4, 5];
13 console.log(a);//1
14 console.log(b);//2
15 console.log(c); //[3,4,5]
16
17 //求数组的最大值
18 console.log(Math.max(10, 30, 50, 15));
19 //max(), 零散形式传参
20 console.log(Math.max.apply(null, [30, 70, 80, 20]));
21 //apply劫持，以数组形式传参
22 console.log(Math.max(...[30, 70, 80, 20]));
23 //扩展运算符
24
25 //sort
26 let studentArr = [
```

```

27     {name: 'AAA',score: 90},
28     {name: 'BBB',score: 80},
29     {name: 'CCC',score: 92},
30 ];
31 //按照成绩进行排序
32 let r = studentArr.sort((a, b) => b.score - a.score);
33     console.log(r);
34
35 //随机排序
36 let r1 = studentArr.sort(() => Math.random() - 0.5);
37     //Math.random() - 0.5 返回大于0或小于0的数
38     console.log(r1);

```

扩展运算符可用于合并两个对象

- 若自定义的属性放在扩展运算符后面，则扩展运算符内部有同名属性会被覆盖
- 若把自定义属性放在扩展运算符前面，则变成了设置新对象的默认属性
- 若扩展运算符的参数是undefined或者null，则这两个值会被忽略，不会报错
- 若扩展运算符的参数对象之中有取值函数get，则这个函数会被执行

```

1 let {a, b, ...c} = {a:1, b:2, num:3, num2:4, num3:5};
2     console.log(c); //{num:3, num2:4, num3:5}

```

2、数组、对象方法拓展：

Array.prototype.sort():

语法：

```

1 arr.sort([compareFunction])

```

- 如果 compareFunction(a, b) 小于 0，那么 a 会被排列到 b 之前；
- 如果 compareFunction(a, b) 等于 0，a 和 b 的相对位置不变。备注：ECMAScript 标准并不保证这一行为，而且也不是所有浏览器都会遵守（例如 Mozilla 在 2003 年之前的版本）；
- 如果 compareFunction(a, b) 大于 0，b 会被排列到 a 之前。
- compareFunction(a, b) 必须总是对相同的输入返回相同的比较结果，否则排序的结果将是不确定的。

所以，比较函数格式如下：

```

1 function compare(a, b) {

```

```

2   if (a < b ) {           // 按某种排序标准进行比较, a 小于 b
3       return -1;
4   }
5   if (a > b ) {
6       return 1;
7   }
8   // a must be equal to b
9   return 0;
10  }

```

Object.is () : 判断两个值是否相等, ===

Object.assign () : 将源对象的所有可枚举属性复制到目标对象

```

1  //对对象进行合并操作
2  let obj1 = {a:1, b:2};
3  let obj2 = {c:3, b:200};
4  let obj3 = {d:400, a:300};
5  Object.assign(obj1, obj2, obj3);
6  console.log(obj1); //{a: 300, b: 200, c: 3, d: 400}
7  console.log(obj2); //{c:3, b:200}
8  console.log(obj3); //{d:400, a:300}

```

Object.create: 方法创建一个新对象, 使用现有的对象来提供新创建的对象的__proto__。

```

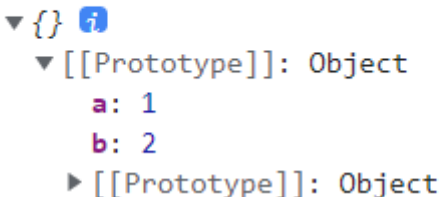
1  //新建一个对象, 参数是他的父亲--->原型  __proto__
2  let obj1 = {a:1, b:2};
3  let newObj = Object.create(obj1);
4  console.log(newObj);

```

```

> let obj1 = {a:1, b:2};
  let newObj = Object.create(obj1);
  console.log(newObj);

```



```

▼ {} ⓘ
  ▼ [[Prototype]]: Object
    a: 1
    b: 2
    ► [[Prototype]]: Object

```

Object.keys (obj) : 返回一个数组，包含对象自身的（不含继承的）所有可枚举属性（不包含symbol属性）

```
1 let obj1 = {a:1, b:2};
2 //数组的keys、values、entries是原型方法，也就是实例化方法，可以直接使用
3 //对象的keys、values、entries是静态方法，
4 console.log(Object.keys(obj1));
5 console.log(Object.values(obj1));
6 console.log(Object.entries(obj1));
```

Object.defineProperty(): 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象。

语法: Object.defineProperty(obj, prop, descriptor)

```
1 let obj = { a: 1 };
2 Object.defineProperty(obj, 'b', { //给obj新定义了一个属性b
3     value:20, //新属性的值
4
5     writable:true, //是否可以修改value值
6
7     configurable:true, //是否可以配置，比如delete
8
9     enumerable:true //for...in是否可以循环到，枚举
10 });
11 console.log(obj); //{a: 1, b: 20}
12
13 obj.b = 300; //要确保writable为true，才能修改属性的值，默认值为false
14 console.log(obj.b); //300
15
16 delete obj.b; //要确保 configurable:true，默认值为false
17
18 //对对象进行遍历，要确保enumerable:true，默认值为false
19 for (let k in obj) {
20     console.log(k); //a
21 }
```

Object.defineProperty.set.get

```

1 let obj = { a: 1 };
2 let bValue = 200;
3 Object.defineProperty(obj, 'b', {
4     get(){
5         console.log('get--->读取属性值会执行');
6         return bValue;
7     },
8     set(newValue){
9         console.log('set--->修改属性值会执行');
10        bValue = newValue;
11    }
12 });
13
14 console.log(obj.b); //200
15 obj.b = 300;
16 console.log(obj.b); //300

```

3、属性的简洁表示法

```

1 <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
2 <script>
3     let username = 'AAA';
4     //let obj = { username: username };
5     let obj = {
6         username
7     }; //和上面是等价的
8     console.log(obj);
9
10    //会把从数据库里面查询到的数据放到是一个数组里面
11    let data = [{}, {}, {}];
12    let result = {
13        code: 1,
14        data,
15        msg: 'success'
16    };
17    console.log(result);
18

```

```

19      //下面两个是等价的
20      let obj1 = {
21          fn: function () {}
22      };
23      let obj2 = {
24          fn() {},
25          a: 200,
26          b: 300,
27          f() {},
28          v: 800
29      };
30
31      //表单里面的数据提交给服务器
32      let tel = document.querySelector('input[name="tel"]').value;
33      let passwd = document.querySelector('input[name="passwd"]').value;
34      //提交给服务器
35      axios.post('/user', {
36          tel,
37          passwd
38      })
39          .then(function (response) {
40              console.log(response);
41          })
42          .catch(function (error) {
43              console.log(error);
44          });
45  </script>

```

4、Symbol:

新的数据类型，表示独一无二的值

ES6 引入了一种新的原始数据类型 **Symbol**，表示独一无二的值。它属于 JavaScript 语言的数据类型之一，其他数据类型是：**undefined**、**null**、布尔值 (Boolean)、字符串 (String)、数值 (Number)、大整数 (BigInt)、对象 (Object)。

```

1  let watchData = Symbol('compile');
2      console.log(watchData); //生成一个独一无二的值
3      console.log(typeof watchData); //symbol
4

```

```
5 let Compile = Symbol('compile');
6 console.log(Compile);
7 console.log(typeof Compile);//symbol
8 console.log(watchData == Compile);//false
```

通过symbol函数生成:

```
1 let s = Symbol();
```

对象属性名的两种表示方法: 1、原有字符串表示 2、Symbol表示

特点:

- Symbol可接受一个字符串作为参数, 表示对Symbol实例的描述, 主要是为了在控制台输出, 或者是转为字符串时比较容易区分
- 如果Symbol的参数是一个对象, 则可以调用toString方法将其转换成为字符串, 然后生成一个Symbol值
- Symbol值不能与其他类型的值进行运算, 否则会报错
- Symbol值可以显式的转为字符串
- Symbol 的值是唯一的, 用来解决命名冲突的问题
- Symbol 定义的对象属性不能使用 for...in 循环遍历, 但是可以使用 Reflect.ownKeys 来获取对象的所有键名

```
1 let sym=Symbol('My Symbol');
2 String(sym); //'Symbol(My Symbol)'
3 sym.toString(); //'Symbol(My Symbol)'
```

Symbol值也可以转换成为布尔值, 但是不能转换为数值

```
1 let sym=Symbol();
2 Boolean(sym); //true
3 !sym;          //false
4 Number(sym); //TypeError
```

Symbol值作为对象的属性名时不能使用点运算符, 只能用方括号

在对象内部使用Symbol值定义属性时, 也只能用方括号

```
1 let sym=Symbol();
2 let a={};
3 a[sym]='Hello';
```

```
4 a[sym]; //undefined
5 s['sym']; //'Hello'
```

Symbol类型的值还可以定义一组常量，保证这些常量都是不相等的
Symbol.for() 可以做到使用同一个Symbol值，

```
1 let s1 = Symbol.for('foo');
2 let s2 = Symbol.for('foo');
3 s1 == s2; // true
```

Symbol内置值：

内置值	描述
Symbol.hasInstance	当其它对象使用 instanceof 运算符，判断是否为该对象的实例时，会调用这个方法
Symbol.isConcatSpreadable	对象的 Symbol.isConcatSpreadable 属性等于的是一个布尔值，表示该对象用于 Array.prototype.concat()时，是否可以展开
Symbol.species	创建衍生对象时，会使用该属性
Symbol.match	当执行 str.match(myObject) 时，如果该属性存在，会调用它，返回该方法的返回值
Symbol.replace	当该对象被 str.replace(myObject)方法调用时，会返回该方法的返回值
Symbol.search	当该对象被 str.search (myObject)方法调用时，会返回该方法的返回值
Symbol.split	当该对象被 str.split(myObject)方法调用时，会返回该方法的返回值
Symbol.iterator	当对象进行 for...of 循环时，会调用 Symbol.iterator 方法，返回该对象的默认遍历器
Symbol.toPrimitive	当对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值
Symbol.toStringTag	当对象上面调用 toString 方法时，返回该方法的返回值
Symbol.unscopables	当对象指定了使用 with 关键字时，哪些属性会被 with 环境排除

5、遍历器

遍历器（Iterator）就是一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作。ES6 创造了一种新的遍历命令 for...of 循环，Iterator 接口主要供 for...of 消费

工作原理：

1. 创建一个指针对象，指向当前数据结构的起始位置
2. 第一次调用对象的 next 方法，指针自动指向数据结构的第一个成员
3. 接下来不断调用 next 方法，指针一直往后移动，直到指向最后一个成员
4. 每调用 next 方法返回一个包含 value 和 done 属性的对象

注意：需要自定义遍历数据的时候，要想到迭代器


```

1 //声明一个数组
2     const xiyou = ["AAA", "BBB", "CCC", "DDD"];
3     //使用 for...of 遍历数组
4     for (let v of xiyou) {
5         console.log(v);
6     }
7     console.log("=====");
8
9     //获取迭代器对象
10    let iterator = xiyou[Symbol.iterator]();
11    //调用对象的next方法
12    console.log(iterator.next());
13    console.log(iterator.next());
14    console.log(iterator.next());
15    console.log(iterator.next());
16    console.log(iterator.next());

```

AAA	遍历器.html:17
BBB	遍历器.html:17
CCC	遍历器.html:17
DDD	遍历器.html:17
=====	遍历器.html:19
▶ {value: 'AAA', done: false}	遍历器.html:24
▶ {value: 'BBB', done: false}	遍历器.html:25
▶ {value: 'CCC', done: false}	遍历器.html:26
▶ {value: 'DDD', done: false}	遍历器.html:27
▶ {value: undefined, done: true}	遍历器.html:28

自定义遍历数据：

```

1 //声明一个对象
2     const banji = {
3         name: "五班",
4         stus: [
5             "张三",
6             "李四",
7             "王五",
8             "小六"
9         ],

```

```

10     [Symbol.iterator]() {
11         //索引变量
12         let index = 0;
13         let _this = this;
14         return {
15             next: function () {
16                 if (index < _this.stus.length) {
17                     const result = {
18                         value: _this.stus[index],
19                         done: false
20                     }; //下标自增 index++; //返回结果
21                     return result;
22                 } else {
23                     return {
24                         value: undefined,
25                         done: true
26                     };
27                 }
28             }
29         };
30     }
31     } //遍历这个对象
32     for (let v of banji) {
33         console.log(v);
34     }

```

6、Set:

类似于数组，其成员唯一，不重复

Set本身是一个构造函数，用于生成Set数据结构

```

1  let set1 = new Set();
2  console.log(set1); //{size: 0}
3  //添加
4  set1.add(100);
5  set1.add(200);
6  set1.add(300);
7  console.log(set1); //{100, 200, 300}
8  console.log(typeof set1); //类型是object

```

```
9 //返回Set实例成员总数
10 console.log(set1.size); //3
11 //删除
12 set1.delete(200);
13 console.log(set1); //{100, 300}
14 //检测
15 console.log(set.has(300)); //true
```

Set函数可以接受一个数组，作为参数，用于初始化

```
1 //把数组转换为set
2 let set2 = new Set([100, 200, 300]);
3 console.log(set2); //{100, 200, 300}
4
5 let arr = [1,2,3,4,32,3,32,6];
6 console.log(Array.from(new Set(arr)));
7 console.log([...new Set(arr)]); //数组去重最简单的方式
```

Set实例的属性：

- Set.prototype.Constructor ()：构造函数，就是Set ()
- Set.prototype.size ()：返回Set实例成员总数
- add (value)：添加值，返回Set本身
- delete (value)：删除值，返回一个布尔值，表示删除是否成功
- has (value)：返回布尔值，表参数是否为Set成员
- clear ()：清除所有成员，无返回值

遍历：

- keys ()：返回键名的遍历器
- values ()：返回键值的遍历器
- entries ()：返回键值对的遍历器
- forEach (function () {})：使用回调函数遍历每个成员，无返回值（可加参2，表示绑定的this对象）

```
1 //用构造函数的原型思想去理解所有的方法
2 let set = new Set();
3 set.add(500);
4 console.log(set);
5
6 console.log(set.has(300)); //false
```

```
7 console.log(set.has(500)); //true
8 console.log(set.keys()); //SetIterator {500}
9 console.log(set.entries()); //SetIterator {500 => 500} 可以观察到键值对是相等的
```

Set.数组的操作:

```
1 let arr1 = [1, 2, 2, 2, 3, 4, 32, 56, 78, 90],
2     arr2 = [2, 2, 33, 5, 89, 90, 23, 45];
3 //1求数组的并集
4 console.log([...new Set([...arr1, ...arr2])]);
5 // [1, 2, 3, 4, 32, 56, 78, 90, 33, 5, 89, 23, 45]
6 //2求数组的交集
7 console.log([...new Set(arr1.filter(n => arr2.includes(n)))]); // [2, 90]
8 console.log([...new Set(arr1)].filter(item => new Set(arr2).has(item)));
9
10 //3求数组的差集
11 var a = new Set([1, 2, 3]);
12 var b = new Set([4, 3, 2]);
13 var difference = new Set([...a].filter(x => !b.has(x)));
14 console.log(difference); //Set(1) {1}
```

Array.from方法可以将 Set 结构转为数组

```
1 const items = new Set([1, 2, 3, 4, 5]);
2 const array = Array.from(items);
3 console.log(array); // [1, 2, 3, 4, 5]
```

Vue简单封装:

html代码:

```
1 <div id="app">我是{{username}}, 今年{{age}}岁了, 性别{{gender}}</div>
2
3 <!-- 引入Vue框架 -->
4 <script src="./js/Vue.js"></script>
5
6 <--- 依赖Vue框架写业务 -->
7 <script src="./js/index.js"></script>
```

index.js代码:

```

1 //使用自运行函数形成封闭的空间
2 (function (){
3     let n=200;
4     let app= new Vue ({
5         el:'#app',
6         data:{
7             username:'AAA',
8             age:19,
9             gender:'male'
10        }
11    })
12    app.$data.username = 'BBB';
13    setTimeout(function(){
14        app.username='CCC';
15        app.age= 20;
16        app.gender = 'female';
17    },5000);
18
19    setTimeout(function(){
20        app.username='DDD';
21        app.age= 21;
22        app.gender = 'male';
23    },5000);
24 })();

```

Vue.js代码:

```

1 (function(){
2     let n =20;
3     //在构造函数里面，基本就是属性的初始化操作
4     //在构造函数里面，很少有具体的功能实现
5     function Vue ({el,data}){
6         this.$el = el;
7
8         this.$data = data;
9         //缓存一些数据
10        this.node = document.querySelector(el);
11

```

```
12      //把节点里的 原始模版 缓存起来
13      this.nodeHTML = document.querySelector(e1).innerHTML;
14
15      //进行模版编译
16      this.Compile();
17
18      //监听数据改变
19      this.watchData();
20  }
21
22  //在原型之上定义方法、实现功能
23  Vue.prototype.watchData = function(){
24      const _this = this;
25
26      Object.keys(_this.$data).forEach(k=>{
27
28          let pValue = _this.$data[k];
29
30          Object.defineProperty(_this.$data,k,{
31              get(){
32                  return pValue;
33              },
34              set(newValue){
35                  console.log('修改了属性值');
36                  pValue = newValue;
37
38                  //重新对模版进行编译
39                  _this.Compile();
40              }
41          });
42
43          //通过this对属性值进行代理
44          Object.defineProperty(_this,k,{
45              get(newValue){
46                  return _this.$data[k];
47              },
48              set(newValue){
49                  _this.$data[k]=newValue;
50              }
51          });
```

```

52     });
53 }
54
55 Vue.prototype.Compile = function(){
56     //拿到所有的模板绑定的数据---->我是{{username}}, 今年{{age}}岁了, 性别{{gender}}
57     let mustCacheArray = this.nodeHTML.split('{{');
58     //通过split()方法截取
59     console.log(mustCacheArray);
60     //----> ['我是', 'username}}, 今年', 'age}}岁了, 性别', 'gender}}']
61
62     //删除第一个多余的元素
63     mustCacheArray.shift(); // 也就是删除 '我是'
64
65     let html = this.nodeHTML;
66     //----> 我是{{username}}, 今年{{age}}岁了, 性别{{gender}}
67
68     mustCacheArray.forEach(p => {
69         //把username age gender遍历出来
70         let k = p.substring(0, p.indexOf('}}'));
71
72         //替换遍历出来的结果
73         html = html.replace(`{{${k}}}`, this.$data[k]);
74     });
75     //把替换后的结果返回页面
76     this.node.innerHTML = html;
77 }
78 //需要Vue暴露出去
79 window.Vue = Vue;
80 })();

```

js静态方法、原型方法和实例方法的主要区别：

类型	说明	访问者
静态方法	定义在构造函数上的方法	只能被构造函数访问
原型方法	在构造函数的prototype原型之上定义的方法	能被实例直接访问，构造函数需通过prototype才可访问
实例方法	构造函数中this上添加的属性都属于实例属性	只能被实例对象访问

```
1 // 构造函数
2 function People(name) {
3     this.name = name;
4     this.username = function () {
5         console.log("name is " + this.name)
6     }
7 }
8
9 // 静态方法
10 People.say = function () {
11     console.log("静态方法name is ", this.name)
12 }
13
14 // 原型方法
15 People.prototype.foo = function () {
16     console.log("原型方法name is ", this.name)
17 }
18
19 // 实例
20 var fn = new People();
21
22 // 调用静态方法
23 People.say(); // 静态方法name is undefined
24 fn.say(); // 报错信息: Uncaught TypeError: fn.say is not a function
25
26 // 调用原型方法
27 People.foo(); // 报错信息: Uncaught TypeError: People.foo is not a function
28 fn.foo(); // 原型方法name is Red
29
30 // 调用实例方法
31 People.username(); // Uncaught TypeError: People.username is not a function
32 fn.username(); // name is Tom
```